1. In this lab, I first implemented a page-level locking scheme. I created a new class called LockManager to be responsible for granting and releasing shared and exclusive locks on each page, and detecting deadlocks by looking for cycles in the wait-for graph and timeouts. In LockManager, there are three fields, shared locks mapping, exclusive locks mapping, and wait-for graph. One of the core functions is acquireLock(tid, pid, perm), which detects deadlocks, grants a shared lock on the page for the transaction if the permission is read-only, and grants an exclusive lock if the permission is read-write. I will discuss deadlock detection in a later part.
   Here, I created two helper methods for granting two different locks, which return a boolean to represent whether the lock is successfully granted. In acquireSharedLock(tid, pid), I first check if the transaction has already held a shared lock on the page. If not, then check if the page is exclusively locked by some transaction. If so, the lock cannot be granted, returning false. If not, the shared lock can be granted by adding the tid to the shared lock mapping, returning true. Similarly, in acquireExclusiveLock(tid, pid), if the transaction already holds the lock, return true. Then, if some other transaction holds the exclusive lock on the page, return false. Then, if no transactions hold shared locks on the page, the exclusive lock is granted by adding the tid to the exclusive lock mapping, return true. After that, if the transaction holds a shared lock on the page and it's the only one, the shared lock is upgraded to the exclusive lock by removing the tid in shared mapping and adding it to the exclusive lock mapping. In other cases, the exclusive lock is not granted.
   Now, go back to acquireLock(tid, pid, perm). If it receives false from the helper methods, that transaction then waits for some timeout (which will be discussed in the design decision part.) After timeout, the transaction releases all locks acquired by it and aborts.
   Another two main functions are releaseAll(tid) and release(tid, pid). ReleaseAll releases all locks held by the transaction, removes the transaction node (and all incident edges) in the wait-for graph, and notifies all waiting threads. Release just releases the lock on the page held by the transaction, and notifies all waiting threads.
   After implementing the LockManager, I turned to modify the BufferPool. I added a LockManager field to organize locks on pages. In getPage(), I called lockManager.acquireLock(tid, pid, perm) on the first line. I put all lines for getting the page and updating the LRU list in a synchronized block to avoid concurrency issues with the list whose implementation is not synchronized. For releasePage(), I called the releaseLock(tid, pid) in the LockManager. For holdLock(), I called the two helper methods in LockManager, holdsSharedLock and holdsExclusiveLock.
   Then, I implemented the No-Steal policy. In evictPage(), instead of evicting the LRU page, I evicted the LRU non-dirty page by traversing the LRU list. If all pages are dirty, an exception is thrown.
   Then, I implemented transactionComplete and the Force policy. In transactionComplete(tid, commit flag), if the transaction is committed, I flushed all pages dirtied by the transaction, or if the transaction is aborted, I discarded all pages dirtied by the transaction by calling discardPage on every page. After these, I called lockManager.releaseAll(tid) which releases all locks by tid and updates the wait-for

graph.
The last thing I implemented is deadlock detection and resolution. I will discuss it later.

2. Add a unit test that a transaction reads a page, then a transaction writes on the page and aborts, and the first transaction reads the page again.
3. Design decision:
   For Locking granularity, I chose the page-level locking granularity. So in LockManager, the shared lock map and exclusive lock map map page id to transaction id.
   Also, I implemented both timeouts and the dependency graph to avoid deadlock situations, and chose to abort the current transaction as deadlock resolution. At the beginning of acquireLock() in LockManager, I first added the transaction to the wait-for graph based on the permission and which other transactions hold conflict locks on the page. Then, I called hasDeadLock(). If it returns true, I called releaseAll(tid) and threw the TransactionAbortException, and the current transaction aborts. The current transaction will also abort if it waits until the timeout. After the timeout, I also called releaseAll(tid) and threw the TransactionAbortException to abort the current transaction.
4. I added a Permission parameter in holdLock(). This function then checks that the transaction holds a shared lock if the permission is read-only and checks that the transaction holds an exclusive lock if the permission is read-write.
5. All completed.